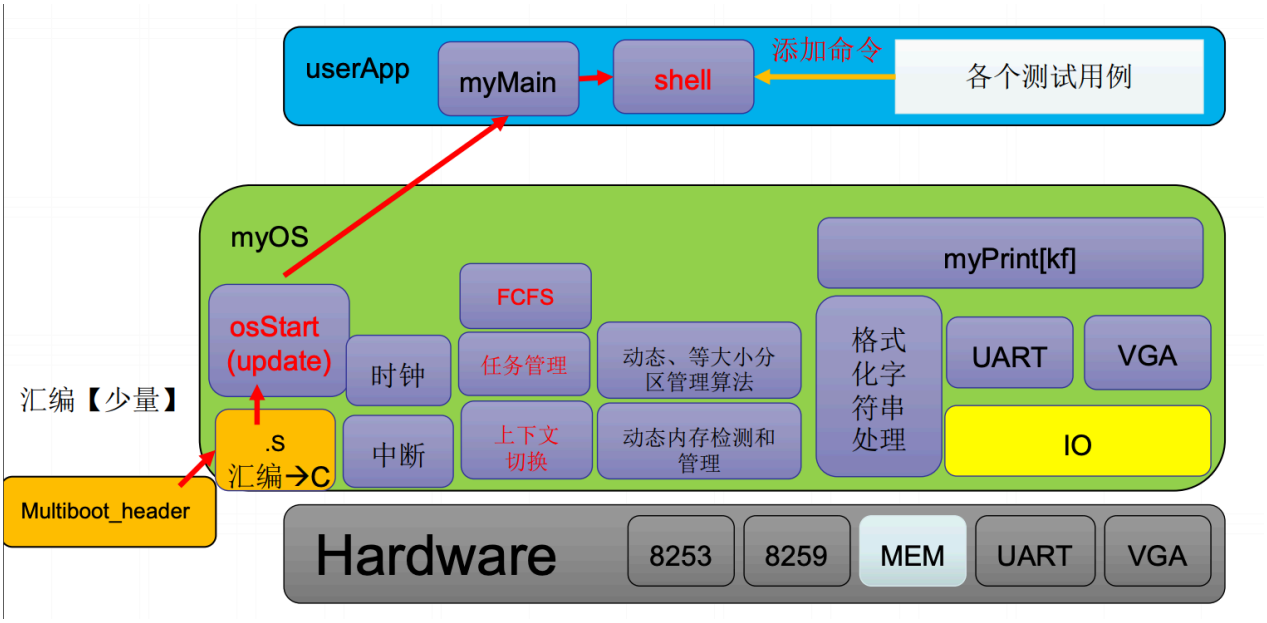
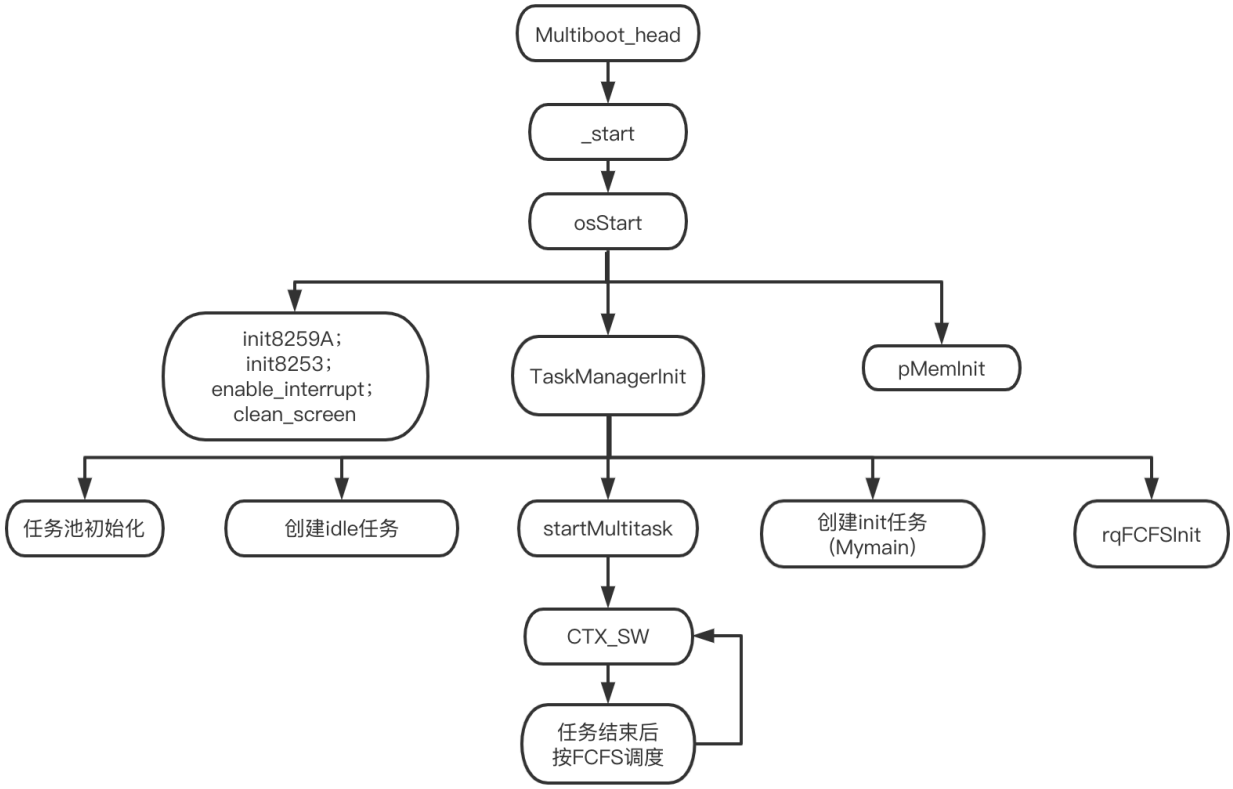
实验五 任务管理器

软件框架及概述



概述：从Multiboot_header进入入操作系统内核(myOS)，为进入C程序准备好上下文，初始化操作系统，新增了上下文切换、任务管理和调度功能。调用myMain(封装成一个任务)进入userApp。userApp中主要实现了shell功能(封装成任务)。

主流程及其实现



主流程图

主流程说明：multiboot_header.s调用_start入口，myOS中的Start32.s设置好中断处理，设置时钟中断(包括后续的tick维护、墙钟维护和显示)，提供了osStart()入口，做好第一次调用C语言入口前的准备，进入osStart()后，先初始化操作系统，包括初始化i8259、i8253，并开中断，检测内存并初始化，最后进行任务管理器初始化，包括任务池初始化，创建idle任务，初始化FCFS调度队列，创建init任务，进入多任务状态。首先开始执行init任务(Mymain)，任务结束就调度出下一个任务继续执行，如此往返。

主要功能模块及其实现&源代码说明

实验5的基础来自助教提供的框架

模块一 任务管理

定义任务数据结构和任务池如下所示，myTCB包括栈顶指针、栈、任务id、状态、调度相关参数next(指向任务池中的后一个任务)。任务池以任务数组形式静态分配。

```
typedef struct myTCB {
     unsigned long * stkTop;     /* 栈顶指针 */
     unsigned long stack[STACK_SIZE];
     int tcbIndex;
     struct myTCB *next;
     int state;
} myTCB;

myTCB tcbPool[TASK_NUM];
```

模块二 任务的创建和销毁

创建任务：选择任务池中下一个空闲TCB，定义好TCB各项参数后启动任务，返回任务id。

销毁任务：任务状态设为-1(表示终结)，准备调度下一个任务。

```
int createTsk(void (*tskBody)(void)) {
     myTCB * allocated = firstFreeTsk;
     if(!allocated) return -1;

     allocated->state = 0;
     stack_init(&(allocated->stkTop), tskBody);
     tskStart(allocated);

     firstFreeTsk = allocated->next;

     return allocated->tcbIndex;
}

void destroyTsk(int takIndex) {
     tcbPool[takIndex].state = -1;

     schedule();
}
```

模块三 任务启动和终止

任务启动：任务状态置为就绪，加入就绪队列。

任务终止：当前任务从就绪队列出列，并销毁。

```
void tskStart(myTCB *tsk){
    tsk->state = TSK_RDY;
    tskEnqueueFCFS(tsk);
}

void tskEnd(void){
    tskDequeueFCFS(currentTsk);
    destroyTsk(currentTsk->tcbIndex);
}
```

模块四 上下文切换

设置好参数调用CTX_SW即可。

```
unsigned long **prevTSK_StackPtr;
unsigned long *nextTSK_StackPtr;
void context_switch(myTCB *prevTsk, myTCB *nextTsk) {
    prevTSK_StackPtr = &prevTsk->stkTop;
    nextTSK_StackPtr = nextTsk->stkTop;

    CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}
```

模块五 调度相关接口

```
void rqFCFSInit(myTCB * idleTsk) {  //初始化就绪队列
    rqFCFS.head = (void*)0;
    rqFCFS.tail = (void*)0;
    rqFCFS.idleTsk = idleTsk;
}

int rqFCFSIsEmpty(void) {           //判断就绪队列是否为空
    return ((rqFCFS.head == (void*)0)&&(rqFCFS.tail == (void*)0));
}

myTCB * nextFCFSTsk(void) {         //返回就绪队列上的下一个任务
    if(rqFCFSIsEmpty())
        return rqFCFS.idleTsk;
    else return rqFCFS.head;
}

void tskEnqueueFCFS(myTCB *tsk) {    //入列
     if(rqFCFSIsEmpty())
          rqFCFS.head = tsk;
     else
          rqFCFS.tail->next = tsk;

     rqFCFS.tail = tsk;
```

```
}

void tskDequeueFCFS(myTCB * tsk) {     //出列
    rqFCFS.head = rqFCFS.head->next;

    if(tsk == rqFCFS.tail)
        rqFCFS.tail = (void*)0;
}

void scheduleFCFS(void) {              //FCFS调度入口
    myTCB * prevTsk = currentTsk;
    currentTsk = nextFCFSTsk();

    context_switch(prevTsk,currentTsk);
}

void schedule(void) {                  //调度入口
     scheduleFCFS();
}
```

模块六 任务管理器初始化

包括任务池初始化，创建idle任务，初始化FCFS调度队列，创建init任务，进入多任务状态。

```
void tskIdleBdy(void) {     //idle任务主体
    while(rqFCFSIsEmpty);
    schedule();
}

void startMultitask(void) {      //进入多任务运行
     BspContext = BspContextBase + STACK_SIZE -1;
     prevTSK_StackPtr = &BspContext;
     currentTsk = nextFCFSTsk();
     nextTSK_StackPtr = currentTsk->stkTop;
     CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}

void TaskManagerInit(void) {      //任务管理器初始化
     int i;
     myTCB *thisTCB;

     for(i=0;i<TASK_NUM;i++){
          thisTCB=&tcbPool[i];

          //init index
          thisTCB->tcbIndex=i;

          //init freelist
          if(i==TASK_NUM-1)
```

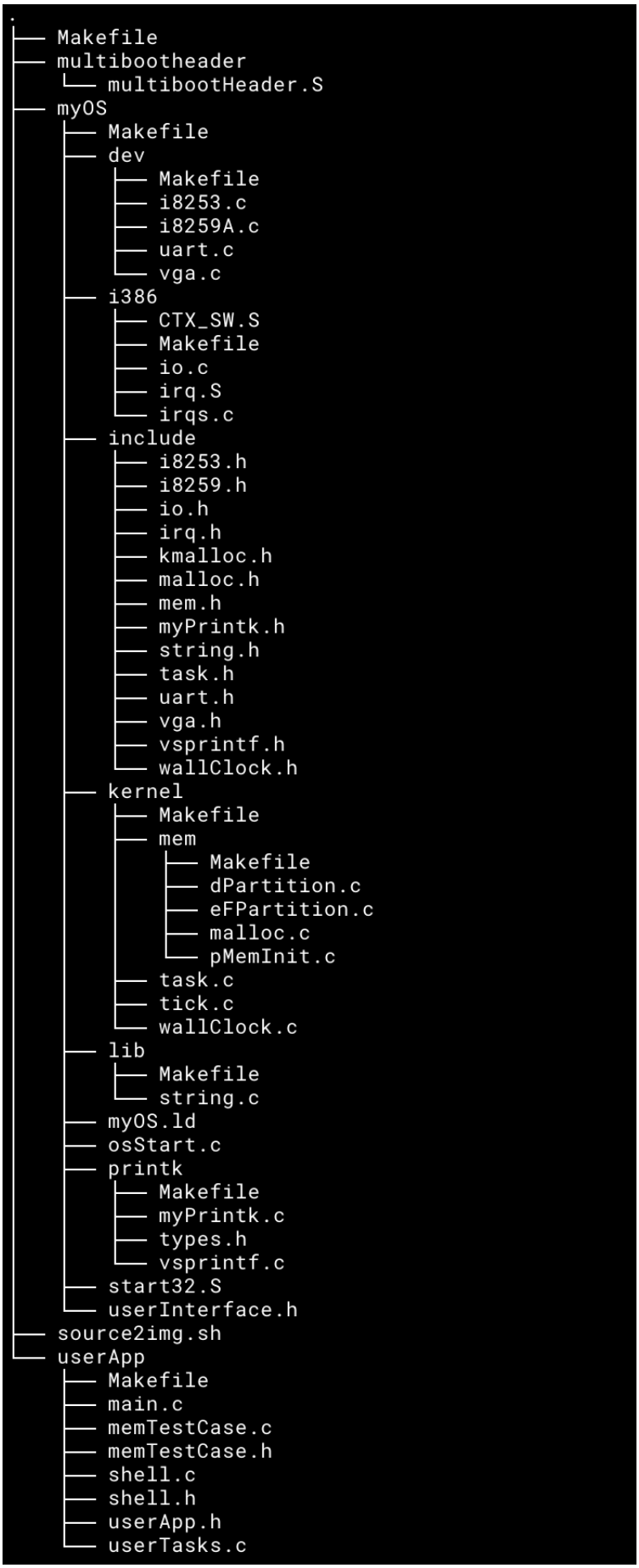
```
                thisTCB->next=(myTCB *)0;
          else
                thisTCB->next=&tcbPool[i+1];

          //init stkTop
          thisTCB->stkTop = thisTCB->stack + STACK_SIZE-1;
     }

     //task0:Idle create and start
     idleTsk = &tcbPool[0];
     stack_init(&(idleTsk->stkTop),tskIdleBdy);
     rqFCFSInit(idleTsk);

     firstFreeTsk = &tcbPool[1];

     //task0:myMain
     createTsk(initTskBody);

     myPrintk(0x2,"START MULTITASKING......\n");
     startMultitask();
     myPrintk(0x2,"STOP MULTITASKING......ShutDown\n");
}

```

osStart中需要调用TaskManagerInit接口，此处不再展示。

init任务的主体即封装后的myMain，如下所示：

```
void myMain(void) {
     clear_screen();

     doSomeTestBefore();

     myPrintf(0x7, "*********************************\n");
     myPrintf(0x7, "*        INIT   INIT !          *\n");
     myPrintf(0x7, "*********************************\n");

     createTsk(myTsk0);
     createTsk(myTsk1);
     createTsk(myTsk2);

     initShell();
     memTestCaseInit();
     createTsk(startShell);

     tskEnd();
}
```

目录组织


.
├── Makefile
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── Makefile
│   ├── dev
│   │   ├── Makefile
│   │   ├── i8253.c
│   │   ├── i8259A.c
│   │   ├── uart.c
│   │   └── vga.c
│   ├── i386
│   │   ├── CTX_SW.S
│   │   ├── Makefile
│   │   ├── io.c
│   │   ├── irq.S
│   │   └── irqs.c
│   ├── include
│   │   ├── i8253.h
│   │   ├── i8259.h
│   │   ├── io.h
│   │   ├── irq.h
│   │   ├── kmalloc.h
│   │   ├── malloc.h
│   │   ├── mem.h
│   │   ├── myPrintk.h
│   │   ├── string.h
│   │   ├── task.h
│   │   ├── uart.h
│   │   ├── vga.h
│   │   ├── vsprintf.h
│   │   └── wallClock.h
│   ├── kernel
│   │   ├── Makefile
│   │   ├── mem
│   │   │   ├── Makefile
│   │   │   ├── dPartition.c
│   │   │   ├── eFPartition.c
│   │   │   ├── malloc.c
│   │   │   └── pMemInit.c
│   │   ├── task.c
│   │   ├── tick.c
│   │   └── wallClock.c
│   ├── lib
│   │   ├── Makefile
│   │   └── string.c
│   ├── myOS.ld
│   ├── osStart.c
│   ├── printk
│   │   ├── Makefile
│   │   ├── myPrintk.c
│   │   ├── types.h
│   │   └── vsprintf.c
│   ├── start32.S
│   └── userInterface.h
├── source2img.sh
└── userApp
    ├── Makefile
    ├── main.c
    ├── memTestCase.c
    ├── memTestCase.h
    ├── shell.c
    ├── shell.h
    ├── userApp.h
    └── userTasks.c

Makefile组织

关键规则：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS}
-o output/myOS.elf

output/%.o : %.S      #所有的.s生成.o
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c      #所有的.c生成.o
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

先由各级子目录下的.c 和.s文件生成.o文件，再将.o文件作为依赖文件，按照myOS.ld规则链接成终极目标文件myOS.elf。

代码布局说明

首先定位到内存中1M地址处。可执行文件的.text段从此处开始。先存放.multiboot_header段[12字节]，往后对齐8字节后，再存放所有输入文件的.text段。 往后对齐16字节，开始存放可执行文件的.data段，即为所有输入文件的.data段。往后对齐16字节，接着存放可执行文件的.bss段，包括所有输入文件中未初始化的全局变量。bss段结束后再向后对齐16字节，此处以_end作为结束标记。往后对齐512字节。

编译过程说明

直接在终端运行./source2run.sh即可。具体过程是先按照makefile内容进行编译链接，编译成功后再执行命令

```
qemu-system-i386 -kernel output/myOS.elf -serial pty &
```

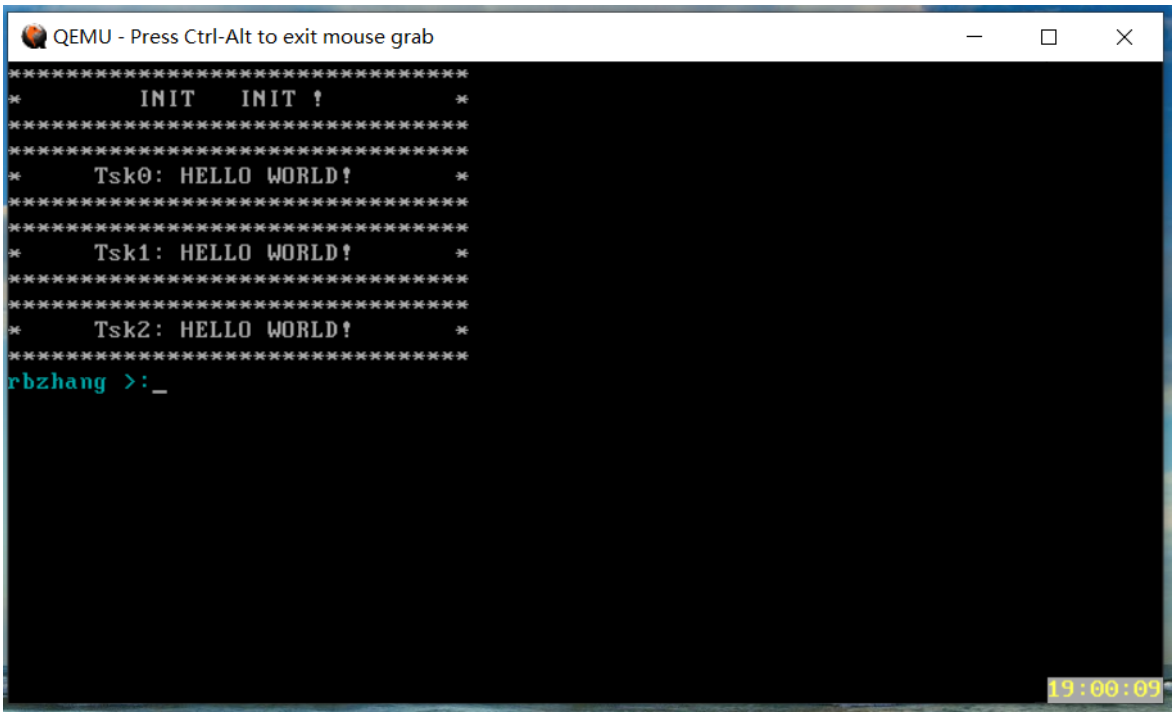
将串口重定向到伪终端，运行时会告知具体是哪个，并据此输入

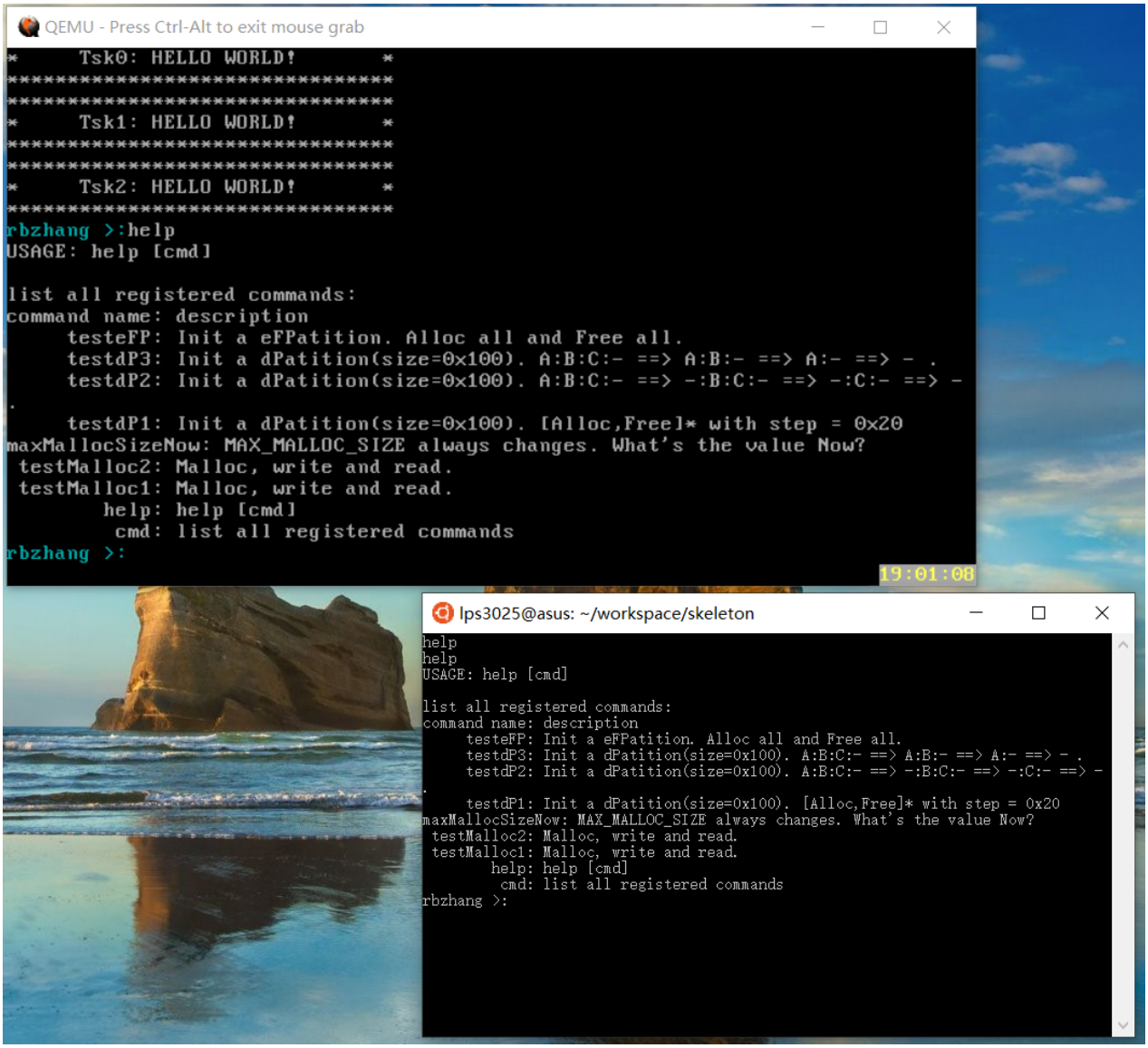
```
sudo screen /dev/pts/0   #假设是/dev/pts/0
```

接着就可以通过伪终端输入命令。

运行和运行结果说明

运行结果如下：





说明：测试用例与老师相同。图一展示了任务调度情况，图二说明shell可正常使用。

遇到的问题和解决方案说明

一开始不太明确调度的时机，最后还是决定放在tskEnd()中调用schedule，即通过一个任务的结束触发调度。对于idle任务，在其内部设置while循环，当就绪队列不为空时(不包括idle自身)，跳出循环开始调度。